Automatic Flow-Control Adaptation for Enhancing Network Performance in Computational Grids *

Wu-chun Feng, Mark K. Gardner and Eric Weigle
({feng,mkg,ehw}@lanl.gov)
Research and Development in Advanced Network Technology (RADIANT)
Advanced Computing Laboratory (ACL)
Computer and Computational Sciences Division
Los Alamos National Laboratory
P.O. Box 1663, M.S. D451
Los Alamos, NM 87545 USA
http://www.lanl.gov/radiant/

Abstract. With the advent of computational grids, networking performance over the wide-area network (WAN) has become a critical component in the grid infrastructure. Unfortunately, many high-performance grid applications only use a small fraction of their available bandwidth because operating systems and their associated protocol stacks are still tuned for yesterday's WAN speeds. As a result, network gurus undertake the tedious process of manually tuning system buffers to allow TCP flow control to scale to today's WAN grid environments. And although recent research has shown how to set the size of these system buffers automatically at connection set-up, the buffer sizes are only appropriate at the beginning of the connection's lifetime. To address these problems, we describe an automated and lightweight technique called Dynamic Right-Sizing that can improve throughput by as much as an order of magnitude while still abiding by TCP semantics.

Keywords: auto-tuning, buffer tuning, Dynamic Right-Sizing, flow-control, TCP.

Nomenclature:
KB – 2^{10} bytes; MB – 2^{20} bytes; Kbps – 10^3 bits per second; Mbps – 10^6 bits per second; Gbps – 10^9 bits per second; PI – protocol interpreter [34].

1. Introduction

TCP has entrenched itself as the ubiquitous transport protocol for the Internet, as well as for emerging infrastructures such as computational grids [16, 17], data grids [3, 9], and access grids [10]. However, parallel and distributed applications running stock TCP implementations perform abysmally over networks with large bandwidth-delay products (BDP) such as are are typical in grid-computing environments and satellite networks [6, 7, 31].

* This work was supported by the U.S. Dept. of Energy through Los Alamos National Laboratory contract W-7405-ENG-36. Any opinions, findings, and conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DOE, Los Alamos National Laboratory.
Los Alamos Unclassified Report (LA-UR) 03-0714

As noted in [6, 7, 12, 31], congestion- and flow-control adaptation bottlenecks, are the primary reason for this abysmal performance. The former is a topic of active research beyond the scope of this paper [8, 15, 22]. In order to address the latter problem, grid and network researchers continue to manually tune buffer sizes to keep the network pipe full [7, 32, 37], and thus achieve acceptable wide-area network (WAN) performance in support of grid computing. However, the tuning process can be quite difficult, particularly for users and developers who are not network experts. It involves calculating the bandwidth of the bottleneck link and the round-trip time (RTT) for a given connection. That is, the optimal TCP buffer size is equal to the product of the bandwidth of the bottleneck link and the RTT, i.e., the effective bandwidth-delay product of the connection.

Currently, in order to tune buffer sizes appropriately, the grid community uses diagnostic tools to determine the RTT and the bandwidth of the bottleneck link. Such tools include *pipechar* [21], *nettimer* [23], *nettest* [24], *pchar* [26], *iperf* [39] and *netspec* [40]. However, none of these tools include a client API so applications can tune their TCP connections and all of the tools require a certain level of network expertise to install and use. Furthermore, many of these tools 'pollute' the network with extraneous probing packets.

To simplify the above tuning process, several services that provide clients with the correct tuning parameters for a given connection have been proposed, e.g., AutoNcFTP [25], Web100 [29] and Enable [38], in order to eliminate what has been called the *wizard gap* [27].[1] Although these services provide good first approximations and can improve overall throughput by two to five times over a stock TCP implementation, they only measure the bandwidth and delay at connection set-up time. This makes the implicit assumption that the bandwidth and RTT of a given connection will not change significantly over the course of the connection. In Section 2, we demonstrate that this assumption is tenuous at best.

A more dynamic approach to optimizing communication in a grid involves automatically tuning buffers over the lifetime of the connection, not just at connection set-up. At present, there exist two kernel-level implementations: auto-tuning [35] and Dynamic Right-Sizing (DRS) [13, 14, 18, 41]. Auto-tuning implements sender-based flow-control adaptation while DRS implements receiver-based flow-control adaptation. Live WAN tests show that DRS in the kernel can achieve a 30-fold increase in throughput when the network is uncongested, although speed-ups of 7-8 times are more typical. Although DRS is fully backwards compatible with regular TCP, achieving large speed-ups requires DRS to be installed on every pair of communicating hosts in a grid. Installing DRS

benefits all TCP-based applications, e.g., *ftp*, multimedia streaming and WWW, not just grid applications.

Installing DRS requires knowledge about modifying, recompiling and installing the kernel, along with root privilege to do so. Thus, DRS functionality is generally not accessible to the typical end user. While we anticipate that DRS will be incorporated into vender's kernels so that it is transparent to the end user, users want improved performance now. Thus, we also propose a more portable implementation of DRS in *user space*. Specifically, we integrate our DRS technique into *ftp* to create drsFTP.

drsFTP is similar in many ways to NLANR's AutoNcFTP [30]. Both are modified FTP implementations which adjust buffer sizes to increase performance. The differences are two-fold. First, AutoNcFTP relies on NcFTP [1] whereas drsFTP uses the de-facto standard FTP daemon originally from Washington University in St. Louis [4] and the open-source Netkit FTP client [2]. Second, the buffers in AutoNcFTP are only tuned at connection set-up while drsFTP buffers are dynamically tuned over the lifetime of the connection to provide better adaptation and better overall performance.

The remainder of the paper is organized as follows. Section 2 demonstrates why dynamic flow-control adaptation is needed over the lifetime of the connection rather than just at connection set-up only. Sections 3 and 4 describe the DRS technique and its implementation in kernel space and in user space, respectively. Then, in Section 5, we present our experimental results, followed by areas of future work in Section 6 and concluding remarks in Section 7.

2. Background

TCP relies on two mechanisms to set its transmission rate: flow control and congestion control. Flow control ensures that the sender does not overrun the receiver's available buffer space (i.e., a sender can send no more data than the size of the receiver's last advertised flow-control window), while congestion control ensures that the sender does not overrun the network's available bandwidth. TCP implements these mechanisms via a flow-control window (*fwnd*) that is advertised by the receiver to the sender and a congestion-control window (*cwnd*) that is adapted by the sender based on the inferred state of the network.[2]

Specifically, TCP calculates an effective window, $ewnd \equiv \min(fwnd, cwnd)$, and then sends data at a rate of $ewnd/RTT$, where RTT is the round-trip time of the connection. Currently, *cwnd* varies dynamically as the network state changes; however, *fwnd* has traditionally

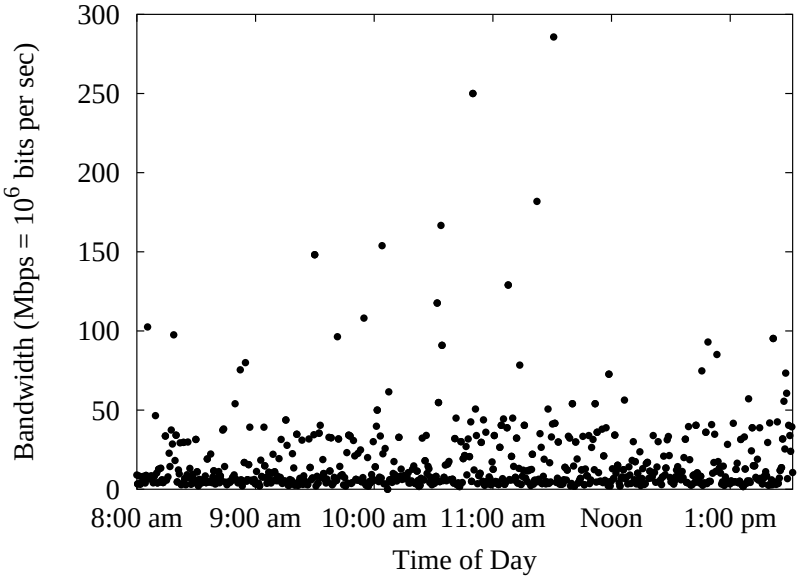

Figure 1. Bottleneck Bandwidth at 20-Second Intervals

been static despite the fact that today's receivers are not nearly as buffer-constrained as they were twenty years ago. Ideally, *fwnd* should vary with the bandwidth-delay product (BDP) of the network, thus providing the motivation for DRS.

Historically, a static *fwnd* sufficed for all communication because the BDP of networks was small. Hence, setting *fwnd* to small values produced acceptable performance while wasting little memory. Today, most operating systems set *fwnd* $\approx 64\,\text{KB}$ — the largest window available without scaling [19]. Yet BDPs range between a few bytes ($56\,\text{Kbps} \times 5\,\text{ms} \rightarrow 36\,\text{bytes}$) and a few megabytes ($622\,\text{Mbps} \times 100\,\text{ms} \rightarrow 7.8\,\text{MB}$). For the former case, the system wastes over 99% of its allocated memory (i.e., $36\,\text{B} / 64\,\text{KB} = 0.05\%$). In the latter case, the system potentially wastes up to 99% of the network bandwidth (i.e., $64\,\text{KB} / 7.8\,\text{MB} = 0.80\%$).

Over the lifetime of a connection, bandwidth and delay change (due to transitory queueing and congestion) implying that the BDP also changes. We use *nettimer* to quantify how much they change. Figures 1, 2, and 3 show that they can vary quite widely.[3] Figure 1 presents the bottleneck bandwidth between Los Alamos and New York at 20-second intervals. The bottleneck bandwidth averages 17.2 Mbps with a low and a high of 0.026 Mbps and 28.5 Mbps, respectively. The standard deviation and half-width of the 95% confidence interval are 26.3 Mbps and 1.8 Mbps. Figure 2 shows the RTT, again between Los Alamos

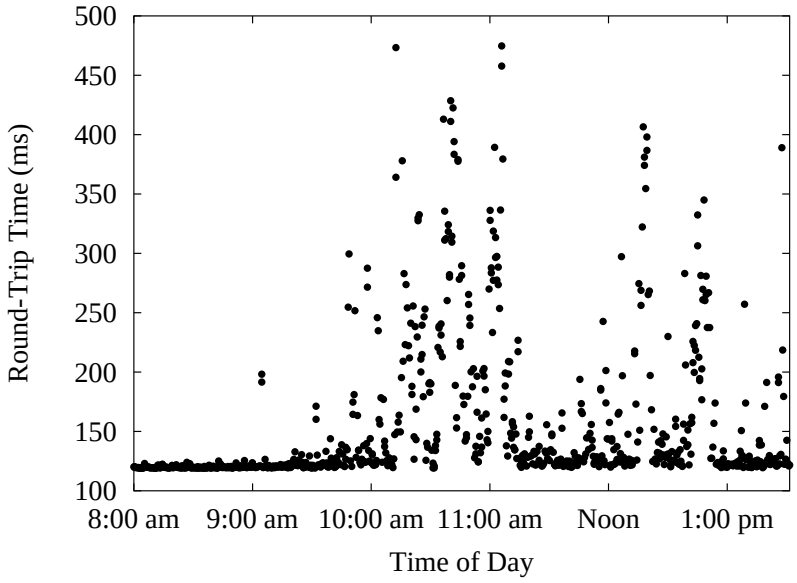

Figure 2. Round-Trip Time at 20-Second Intervals

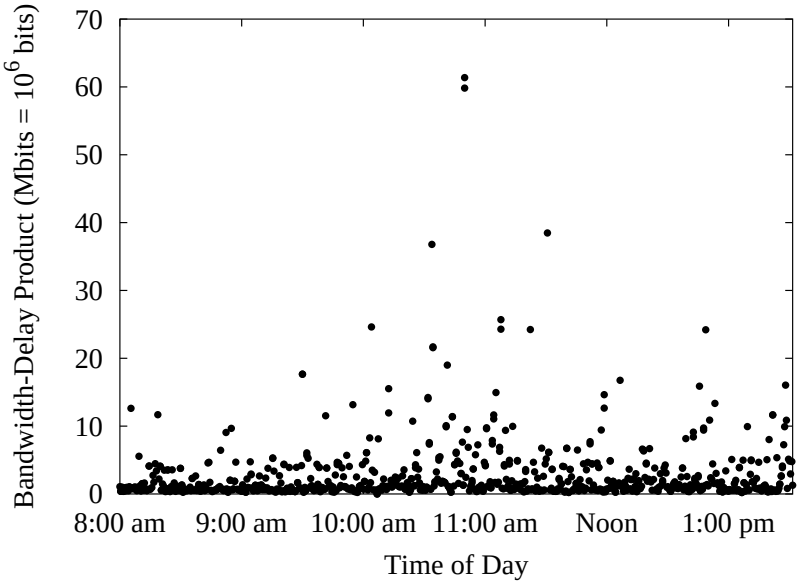

Figure 3. Bandwidth-Delay Product at 20-Second Intervals

and New York, at 20-second intervals. The RTT delay also varies over a wide range 119–475 ms with an average delay of 157 ms. Combining Figures 1 and 2 results in Figure 3, which shows that the BDP of a given connection can vary by as much as 61 Mbit.

Based on the above results, the BDP over the lifetime of a connection is continually changing. Therefore, a fixed value for *fwnd* is not ideal; selecting a fixed value forces an implicit decision between (1) under-allocating memory and under-utilizing the network or (2) over-allocating memory and wasting system resources. Clearly, the grid community needs a solution that dynamically and transparently adapts *fwnd* to achieve good performance without wasting network or memory resources.

3. Dynamic Right-Sizing (DRS) in the Kernel

Dynamic right-sizing (DRS) lets the receiver estimate the sender's *cwnd* and use that estimate to dynamically change the size of the receiver's advertised window *fwnd* as the receiver's memory resources allow. The estimates are also used to keep pace with the growth in the sender's congestion window. As a result, the throughput between end hosts, e.g., in a grid, will only be constrained by the available bandwidth of the network, rather than some arbitrarily set constant value on the receiver.

Initially, at connection set-up, the sender's *cwnd* is smaller than the receiver's advertised window *fwnd*. To ensure that a given connection is not flow-control constrained, the receiver must continue to advertise a *fwnd* that is larger than the sender's *cwnd* before the receiver's next adjustment.

The instantaneous throughput seen by a receiver may be larger than the available end-to-end bandwidth. For instance, data may travel across a slow link only to be queued up on a downstream router and then sent to the receiver in one or more fast bursts. The maximum size of such a burst is bounded by the size of the sender's *cwnd* and the window advertised by the receiver. Because the sender can send no more than one *ewnd* window's worth of data between acknowledgements, a burst that is shorter than a RTT can contain at most one *ewnd*'s worth of data. Thus, for any period of time that is shorter than a RTT, the amount of data seen over that period is a *lower bound* on the size of the sender's *cwnd*. But how does such a distributed system calculate its RTT?

In a typical TCP implementation, the RTT is estimated by observing the time between when data is sent and an acknowledgement is returned. However, during a bulk-data transfer (e.g., from sender to

receiver), the receiver may not be sending any data, and therefore, will not have an accurate RTT estimate. So, how does the receiver infer delay (and bandwidth) when it only has acknowledgements to transmit back and no data to send?

A receiver in a computational grid that is only transmitting acknowledgements can still estimate the RTT by observing the time between when a byte is first acknowledged and the receipt of data that is at least one window beyond the sequence number that was acknowledged. If the sending application does not have any data to transmit, the estimated RTT could be much larger than the actual RTT. Thus, the estimate acts as an *upper bound* on the RTT and should only be used when there is no other source of RTT information. (For a rigorous presentation of the lower and upper bounds, please see [13, 14].)

We note that DRS is also TCP-friendly in the sense that N flows, DRS-enabled or not, will each receive a long-term average of $1/N$-th of the bandwidth of a fully utilized network. Since the congestion-control mechanism governs fairness[4] and because it has the same congestion-control mechanism, DRS responds to congestion the same way as regular TCP. On an uncongested network, however, DRS will attempt to utilize the excess capacity that can exist when all the other connections are artificially limited by their congestion windows. As the network becomes congested again, DRS throttles back and performs no better (or worse) than regular TCP.

It has been suggested that DRS violates TCP semantics by assuming unlimited send buffers. (Large advertised windows require large send buffers if the network is to be kept full.) This is not the case. There is nothing in the TCP specification that requires the sender to allocate buffer space commensurate with the advertised window. A sender is always free to send at a slower rate than the receiver requests, especially if insufficient buffers space is available.

4. DRS in User Space: drsFTP

Unlike the kernel-space version of DRS which benefits all applications transparently, user-space DRS must be implemented by each pair of communicating applications. In this section, we implement DRS in a FTP client and server, resulting in drsFTP.

The primary difficulty in developing user-space DRS applications lies in the fact that user-space code does not have direct access to the state of the TCP stack. Consequently, drsFTP has no knowledge of TCP parameters, such as the RTT of a connection, the receiver's advertised window or the sender's congestion window. Information about a connec-

tion must be estimated from coarse-grained user-space measurements rather than from fine-grained TCP connection state.

FTP specifies that commands and replies are sent over a control channel that is a completely separate TCP connection from the data channel where the transfer take place. As with AutoNcFTP and Enable, we focus on (1) adjusting TCP's system buffers over the data channel of FTP and (2) using FTP's stream file-transfer mode. The latter means that a separate data connection is created for every file transferred. Thus, the sender *always* has data to transmit during the lifetime of the transfer;[5] once the file has been completely sent, the data connection closes

4.1. Determining Available Bandwidth

By definition, we know that the sender always has data to send throughout the life of the FTP data connection. It then follows that the sender will send data as fast as possible, limited by its idea of the congestion- and flow-control windows. Furthermore, the receiver is receiving data as quickly as the current windows, network and CPU scheduling conditions allow. Therefore, the average bandwidth a connection obtains is computed by dividing the number of bytes transmitted by the time required to transmit them.

The difficulty lies in selecting the appropriate sampling interval over which to aggregate the number of bytes transmitted.[6] Selecting too short of an interval dramatically increases overhead and reduces performance. It also leads to erroneous estimates because of scheduling and buffering effects. On the other hand, selecting too long of an interval decreases the responsiveness of DRS to changes in available bandwidth and may reduce performance because the estimated bandwidth-delay product, and hence, the receiver's advertised window, may be artificially small.

In the current implementation of drsFTP, the available bandwidth is computed through the periodic invocation of a signal handler upon alarm expiration. Different values for the sampling interval can easily be tested by varying the expiration time of the alarm. The average bandwidth available to the connection over the last interval is the number of bytes received since the last alarm signal divided by the length of the interval. An appropriate choice for the sample interval yields estimated bandwidth values of sufficient accuracy.

4.2. Determining RTT

Unlike the procedure for estimating the bandwidth of a connection, the RTT cannot be inferred in user-space applications without injecting

a very small amount of extra traffic into the network. User-space code does not have access to the inner workings of the TCP stack and hence cannot know when a given packet is sent nor when its acknowledgement is received.

To sidestep this problem, we send a small packet on the FTP control channel for the sender to echo back. The estimated RTT begins with the sending of a RTT probe packet and ends when its echo is received. The additional load on the network as the result of RTT probe packets is generally small, depending on the sampling interval. (Section 4.4 gives an optimization which minimizes the impact of RTT probes.)

We note that sending the RTT probe packet over the control channel assumes that the control and data channels follow the same route. In the case of third-party control of a FTP data transfer, however, the control and data channels are likely to take very different routes. Thus the RTT estimate may be inaccurate. We send RTT probes over the control channel to comply with RFC 959 [34], since commands cannot be sent on the data channel. If probes could be sent out-of-band on the data channel, then RTT estimates could be obtained in the manner described above. Sending data out-of-band is possible within Globus and hence we are working to integrate drsFTP with GridFTP.

4.3. Adjusting the Receiver's Advertised Window

User-space applications cannot directly set the flow-control window in most TCP stacks. Instead, they must indirectly set the window by setting the TCP receive buffer size to an appropriate value via a `setsockopt` call.

In the worst case, the sender's window is doubling with every round trip during TCP slow start. When it is determined that the receiver window should increase, the new value should be at least double the current value. There is no need to double the current value once TCP is out of slow start. However, it is very difficult, in general, to determine when slow start ends. Therefore, we increase the receive buffer in drsFTP by a factor of two over BDP whenever the current buffer size is less than twice the BDP. (In most protocol stacks, buffer space is not allocated until it is actually used so excessive memory usage is not usually a problem in practice.)

4.4. Adjusting the Sender's Window

To take full advantage of dynamically changing buffer sizes, the sender's buffer should adjust in step with the receiver's. This presents a problem in user-space implementations because the sender's code has no way of determining the receiver's advertised window size. The FTP

protocol specification [34] does not prohibit traffic on the control channel during data transfer, however. Thus, a drsFTP receiver may inform a drsFTP sender about changes in buffer size by sending appropriate messages over the control channel.

Since FTP is a bidirectional data-transfer protocol, the receiver may be either the server or client. RFC 959 specifies that only clients may send commands on the control channel, while servers may only send replies to commands. Thus, a new command and reply must be added in order to fully implement drsFTP. Serendipitously, the Internet Draft of the GridFTP protocol extensions to FTP [5] defines a 'SBUF' command, which is designed to allow a client to set the server's TCP buffer sizes before data transfer commences. We extend the definition of SBUF to allow this command to be specified during a data transfer, i.e., to allow buffer sizes to be set dynamically. The full definition of the expanded SBUF command appears below:

Syntax:

```
sbuf = SBUF <SP> <ID> <SP> <size>
  <ID> ::= <number>
  <size> ::= <number>
```

This command requests the server-PI to set the send-buffer size to `<size>` bytes, assuming sufficient buffer space is available. `<ID>` is provide to match a `SBUF` command to its reply. `SBUF` may be issued at any time, including before or during an active data transfer. If specified during a data transfer, it affects the data transfer that started most recently. The command is informational and need not be acted upon, thus providing interoperability with existing, non-drsFTP, applications.

Response Codes:

```
200 SBUF <SP> <ID> <SP> <size>
```

The server-PI issues a 200 response code containing the `<ID>` of the corresponding command and the new size of the server's buffer. `<ID>` allows the client-PI to match replies to commands in case multiple `SBUF` commands are outstanding in the active transfer. `<size>` allows the client-PI adjust its buffer usage in case the server-PI chooses to allocate less than the requested amount of buffer space.

In addition, we propose a new response code to allow the server-as-receiver to notify the client-as-sender of changes in the receive window.

New Response Code:

> `126 SBUF <SP> <ID> <SP> <size>`
>
> A 126 response may be sent by the server-PI while it is receiving data from the client-PI. As with the `SBUF` command, this reply is informational and need not be acted upon or responded to in any manner by the client-PI. A non-drsFTP application will simply ignore the reply, guaranteeing interoperability with a drs-FTP server.

This response code is consistent with RFC 959 and does not interfere with any FTP extension or proposed extension.

We note that the SBUF command also provides a vehicle for determining RTT without injecting a separate message into the network. Since RTT probes need only contain an `<ID>`, we allow SBUF commands to serve the dual purpose of conveying the receiver's buffer size to the sender and probing for the RTT. Separate RTT probes, as discussed in Section 4.2, are not needed in most instances. Separate probes only become necessary if the time between buffer-size changes becomes so large that the RTT becomes too stale. Since the mechanism for determining RTT via `SBUF` messages is already in place, "empty" `SBUF` messages with the current buffer size serve as the RTT probe in this case.

4.5. TCP Window Scaling

Because the window-scaling factor in TCP is established at connection set-up time, an appropriate scale must be set before a new data connection is opened. Most operating systems allow `TCP_RCVBUF` and `TCP_SNDBUF` to be set on a socket before a connection attempt is made and then use the requested buffer size to establish the TCP window scaling. Figure 4 shows example code for setting an appropriate TCP window scaling factor under most operating systems.

drsFTP sets the send- and receive-buffer sizes to allow windows of up to 16 MB worth of data before initiating connection set-up. Once the connection has been made (and the window scale factor set properly), drsFTP resets the buffer sizes back to their initial values.

In order to set the window scale factor appropriately, the network buffer-size limits of the operating system may need to be increased. The steps involved in increasing the limits are operating system dependent. See [33] for an example of the steps required for a variety of operating systems.

```
/* Set sizes to accommodate window scaling */
byteSize = ftpData->maxBufferSize;
setsockopt(servSock, SOL_SOCKET, SO_RCVBUF, &byteSize, sizeof(int));
setsockopt(servSock, SOL_SOCKET, SO_SNDBUF, &byteSize, sizeof(int));

/* Open the connection */
listen(servSock, SERVER_MAX_CONNECTIONS);
clientSock = accept(servSock, NULL, 0);

/* Set buffer size to appropriate value */
byteSize = ftpData->minBufferSize;
setsockopt(clientSock, SOL_SOCKET, SO_RCVBUF, &byteSize, sizeof(int));
setsockopt(clientSock, SOL_SOCKET, SO_SNDBUF, &byteSize, sizeof(int));
```

Figure 4. Setting TCP Window Scaling in User Space

5. Experiments

In this section, we present results for both kernel- and user-space implementations of DRS. In particular, we will show that the throughput for both the kernel- and user-space implementations improves upon the default configuration by 600% and 300%, respectively.

5.1. Experimental Setup

Our experimental apparatus, shown in Figure 5, consists of three identical machines connected via Fast Ethernet (100 Mbps). The machines need only be fast enough to ensure that the hosts are not the bottleneck. Each machine contains dual 400 MHz Pentium II processors with 128 MB of RAM and two network-interface cards (NICs). One machine acts as a WAN emulator with a 100 ms round-trip time (RTT) delay; each of its NICs is connected to one of the other machines via crossover cables (i.e., no switch).

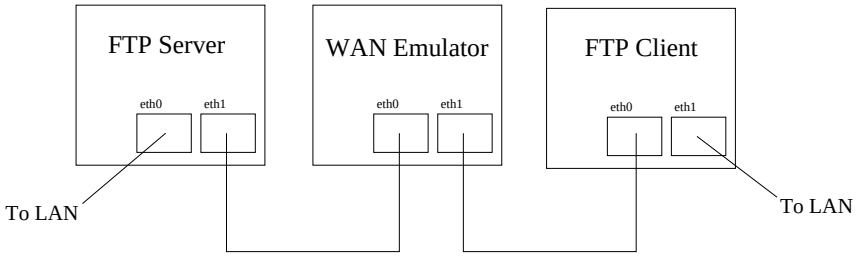

Figure 5. Experimental Apparatus

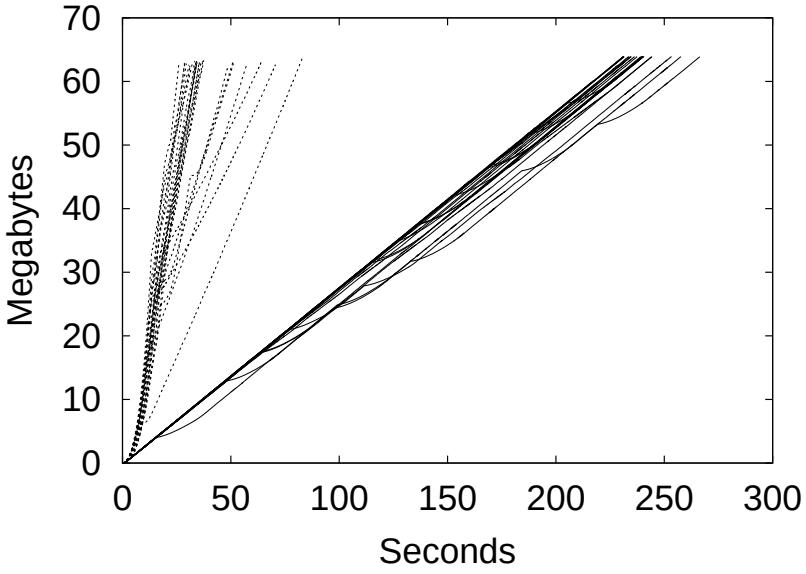

Figure 6. Progress of Data Transfers

5.2. Kernel-Space DRS

In the kernel implementation of DRS, the receiver estimates the size of the sender's congestion window so it can advertise an appropriate flow-control window to the sender. Our experiments show that the DRS algorithm approximates the actual size quite well. Further, we show that by setting the advertised window using this estimate, DRS keeps the connection responsive to congestion while removing the artificial constraints of traditional flow-control adaptation.

5.2.1. *Performance*

As expected, using large flow-control windows significantly enhances WAN throughput versus using the default window sizes of TCP. Figure 6 shows the results of 50 transfers of 64 MB each using `ttcp` [28], 25 transfers with a default window size of 32 KB for both the sender and receiver and 25 transfers with DRS. Transfers with the default window sizes took a median time of 240 seconds to complete while the DRS transfers only took 34 seconds (or roughly *seven* times faster).

Figures 7 and 8 trace the window size and flight size of TCP with the static (default) buffer size and with DRS. (The flight size refers to the amount of sent but unacknowledged data in the sender's buffer. This flight size, in turn, is bounded by the window advertised by the receiver.) For the traditionally static (default) flow-control window as

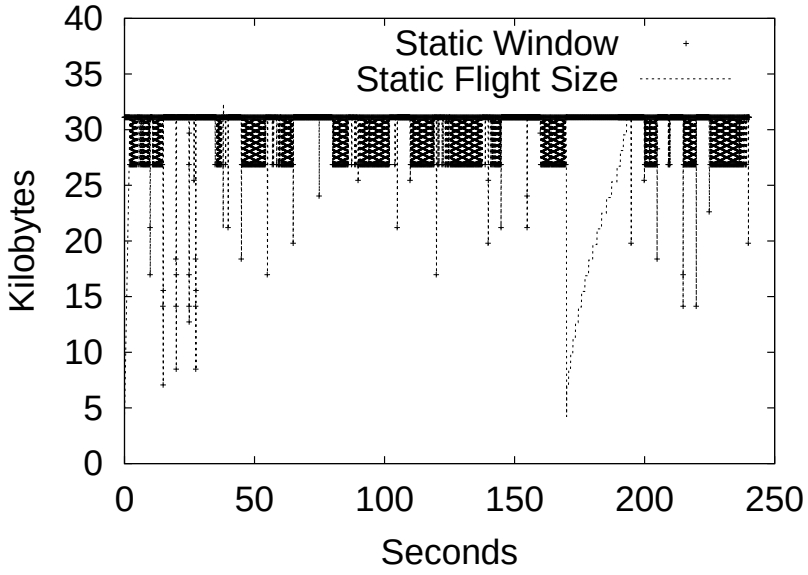

Figure 7. Window and Flight Sizes for Default Buffer Size: Fast Ethernet

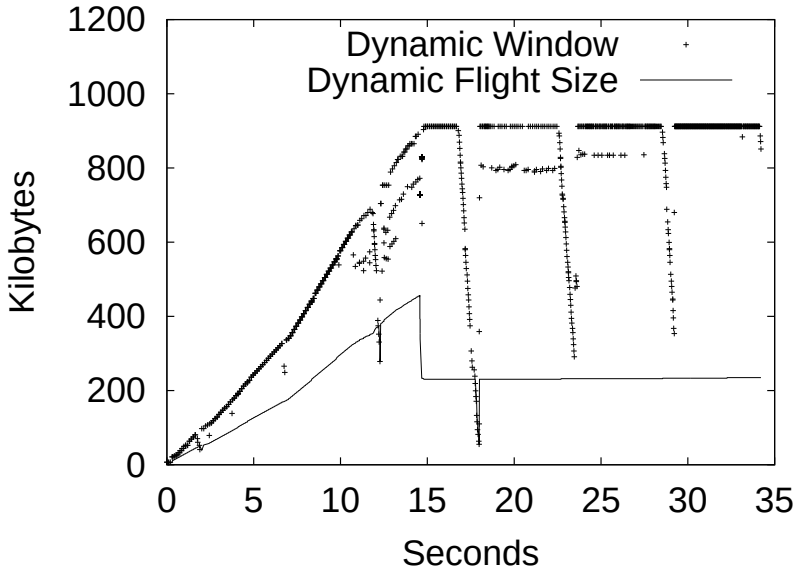

Figure 8. Window and Flight Sizes for Dynamic Right-Sizing: Fast Ethernet

shown in Figure 7, the congestion window quickly grows and the transfer rate is limited by the receiver's small 32 KB window advertisement. On the other hand, DRS allows the receiver to advertise a window size that is roughly twice the largest flight size seen to date (in case the connection is in slow start). Thus, the flight size is only constrained by the conditions in the network, i.e., the congestion window. Slow start continues for much longer and only stops when packet loss occurs. At this point, the congestion window stabilizes on a flight size that is roughly seven times higher than the flight size of the static case. Not coincidentally, this seven-fold increase in the average flight size translates into the same seven-fold increase in throughput shown in Figure 6.

In additional tests, we occasionally observe increased queueing delay caused by the congestion window growing larger than the available bandwidth. This causes the retransmit timer to expire and reset the congestion window to one even though the original transmission of the packet was acknowledged shortly thereafter.

5.2.2. *Parallel Streams*

One approach to achieving a larger (effective) flow-control window is to use parallel streams [32, 36, 37]. We compare the aggregate bandwidth achieved by parallel streams over Linux 2.2.20 (without auto-tuning), Linux 2.4.17 (with auto-tuning) and DRS. The experimental apparatus is similar to that used in Section 5.1 except the hosts have dual 933 MHz Pentium III processors with 512 MB of RAM and Gigabit Ethernet. The round-trip time is 100 ms. (For more information, see [42].)

Figure 9 shows the combined bandwidth of the parallel streams as a function of the number of streams, N. In the first two cases, 2.2.20 (without auto-tuning) and 2.4.17 (with auto-tuning), the combined bandwidth of N streams increases linearly because the aggregate size of the flow-control windows is still less than the congestion-control window.

Since DRS ensures that connections are congestion-control limited rather than flow-control limited, using multiple streams increases the effective rate at which the congestion-control window opens during slow start. Hence the combined bandwidth increases at a faster rate than without DRS.

As the number of parallel streams increases, however, the duration of a transfer decreases for a fixed-size file. Thus the fraction of time spent in slow start increases and the average bandwidth of each stream is reduced. The combination of the two effects causes the combined bandwidth to level off.

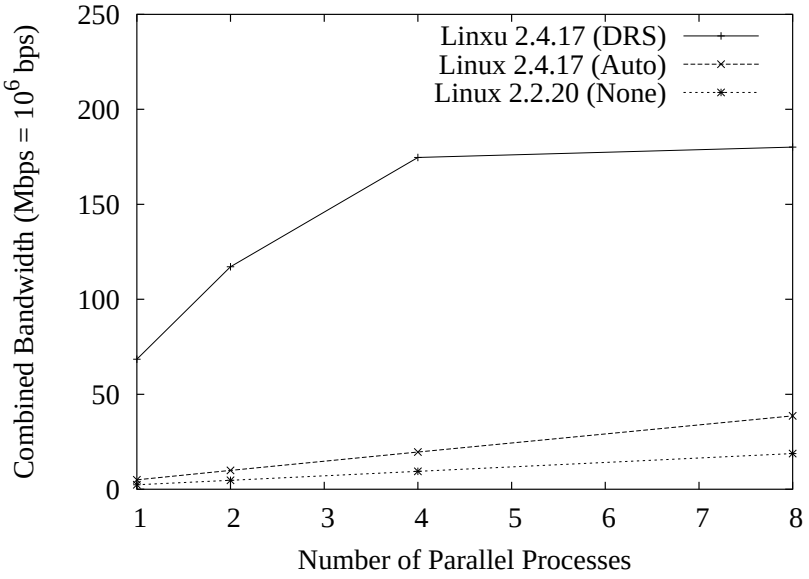

Figure 9. Comparison of Combined Performance with Parallel Streams

5.2.3. *Low-Bandwidth Connections*

Figures 10 and 11 trace the window size and flight size of TCP with the static (default) buffers and with DRS over a 56K modem. Because DRS provides the sender with indirect feedback about the achieved throughput rate, DRS actually causes a TCP Reno sender to induce *less* congestion and fewer retransmissions over bandwidth-limited connections. Although the overall throughput measurements for both cases are virtually identical, the static (default) window generally has more data in flight as evidenced by the roughly 20% increase in the number of re-transmissions shown in Figure 12. This additional data in flight is simply dropped because the link cannot support that throughput.

5.2.4. *Buffer Management*

Under normal circumstances, the Linux 2.4 kernel restricts each connection's send buffers to be just large enough to fill the current congestion window. When the total memory used exceeds a threshold, the memory used by each connection is further constrained.

While DRS was designed to support distributed computing with comparatively few connections over high BDP networks, Linux auto-tuning was designed to support web servers with many concurrent connections over low BDP networks. The two approaches are complimentary; they each handle different ends of the networking spectrum.

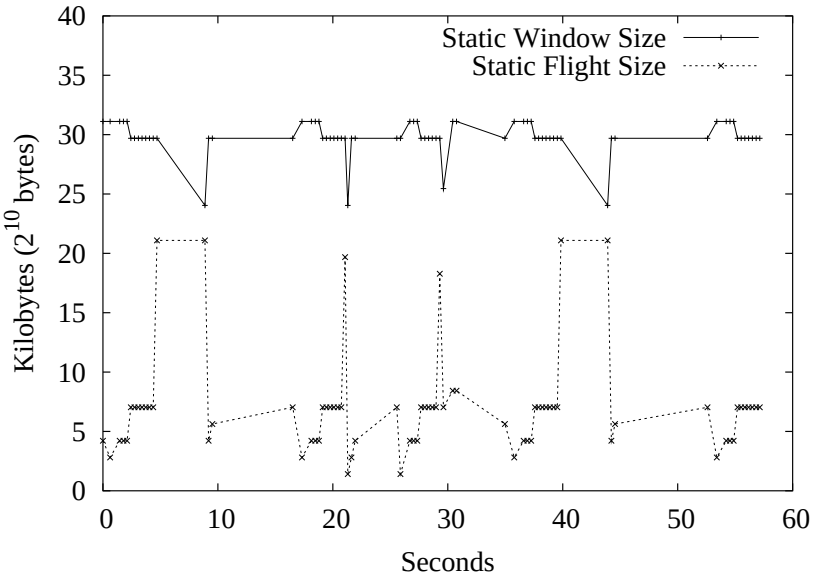

Figure 10. Window and Flight Sizes for Default Buffer Size: 56K Modem

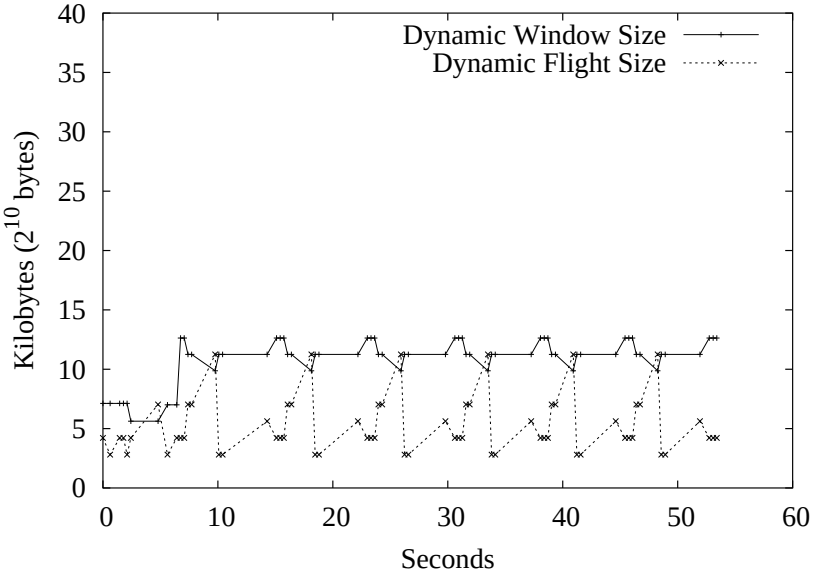

Figure 11. Window and Flight Sizes for Dynamic Right-Sizing: 56K Modem

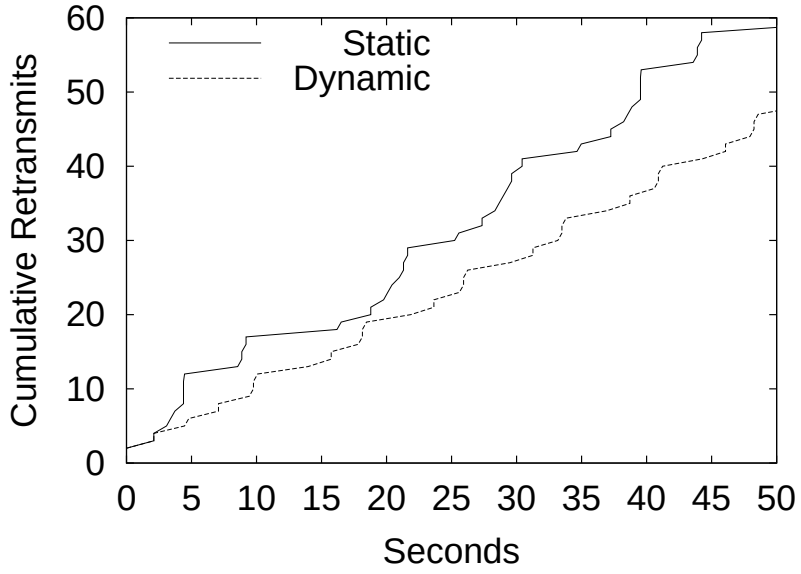

Figure 12. Retransmissions in Low-Bandwidth Links

(A more in-depth analytic and quantitative discussion of the differences between Linux 2.4.x auto-tuning and DRS can be found in [42].)

5.3. drsFTP: DRS in User Space

Since a user-space implementation does not have access to fine-grained TCP state, performance improvements are more modest than for the kernel implementation. Still, the performance is dramatically better than with the default TCP buffer sizes.

5.3.1. *Performance*

The experimental apparatus consists of the same three machines discussed in Section 5.2.2. The WAN emulator, which is implemented using TICKET technology [43], forwards packets at line rate and has a user-settable delay. (In the results that follow, the average round-trip time is 102.1 ms.) All FTP traffic, both data and control, occurs through the WAN emulator.

As a baseline, we use stock FTP with TCP receive buffers set at 64 KB. (Most modern operating systems set their default TCP buffers to 32 or 64 KB. Therefore, this number represents the high end of OS-default TCP buffer sizes.) We next test drsFTP, allowing the buffer size to vary in response to network conditions, starting from 64 KB. Last of all, we test statically-tuned FTP with TCP buffers sizes which represent over- and under-provisioning.

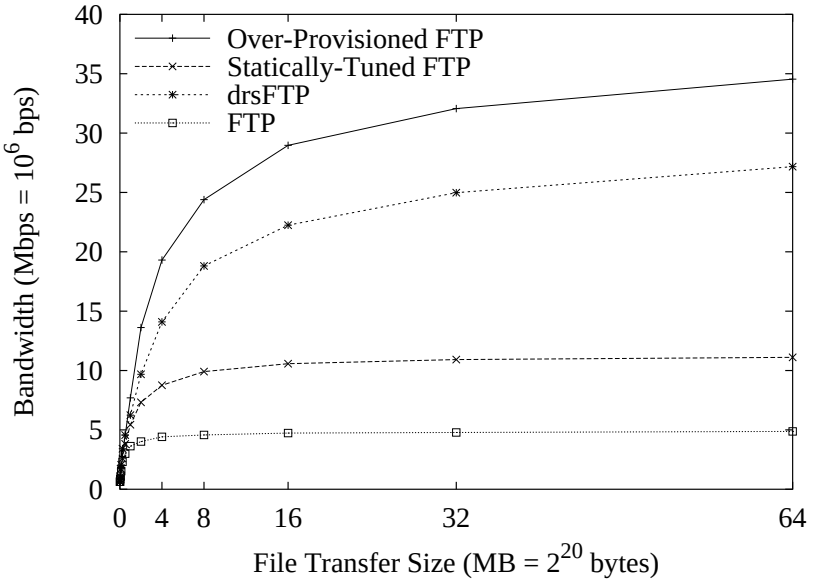

Figure 13. Comparison of FTP, drsFTP and statically-tuned FTP

The over-provisioned buffer size, representing the best performance possible, is 16 MB, which is larger than the BDP (12.2 MB). The under-provisioned buffer size is 212.5 KB, which represents a BDP that is sampled when the network is loaded. (The median value of BDP for the data in Figure 3 is 143.3 KB. A buffer size of 212.5 KB is in the 66th percentile.)

For each test, we transfer a set of files, ranging from 8 KB to 64 MB, over the emulated WAN. The drsFTP sampling interval used to estimate the available bandwidth is one second, a conservative configuration with very low overhead. (The performance is not sensitive to the duration of the sampling interval as long as the sampling interval is greater than the round-trip time. This is an artifact of not emulating cross-traffic.)

Figure 13 shows the average FTP bandwidth as a function of the size of the transfer. (The x-axis has a logarithmic scale with markers placed according to the powers-of-two file sizes tested. The width of the 95% confidence interval is less than $\pm 5\%$ in all cases.) The average bandwidth of FTP with stock buffer sizes approaches 5 Mbps for file sizes as small as 8 MB. In contrast, the average bandwidth of drsFTP asymptotically approaches 30 Mbps at over 64 MB file transfers. Thus, the utilization of available bandwidth by drsFTP is approximately six times better than stock FTP.

The best bandwidth (34.5 Mbps) is achieved by the over-provisioned FTP which has larger-than-required buffer sizes. As shown, drsFTP achieves 78.7% of the over-provisioned bandwidth. The primary reason for the difference in performance is that drsFTP must rely on coarse-grained measurements to infer available bandwidth and round-trip time and hence may not grow the buffer sizes quickly enough. This is an inherent limitation indicative of the interim nature of the drsFTP application. Even though its performance is not as good as the kernel-space implementation [13, 14], drsFTP was developed to provide the benefits of DRS to the grid community while vendors implement DRS in their kernels.

Figure 13 also compares the average bandwidth of drsFTP to a statically-tuned case where the BDP was sampled at an inopportune time, e.g., at one of the lower data points in Figure 3. Here we see that drsFTP utilizes the available bandwidth 2.4 times better than the statically-tuned case. The comparison illustrates the benefit of inferring the available bandwidth and setting the flow-control buffers automatically.

So far, we have only addressed the issue of optimizing transfer rates. We now turn our attention to buffer usage. As motivation, we conjecture that memory consumption will become a more serious issue as computational grids become widely used and hence indispensable parts of the computational infrastructure.

While applications are able to use buffer space with abandon now, we envision the time when grid nodes will become heavily loaded with large numbers of potentially diverse applications. One example might be a repository for human genome information which will be accessed simultaneously by thousands of researchers. If each connection over-provisions its buffers, it is likely that the node will run out of buffer space and reject connections which could otherwise be serviced had the connections been more frugal.

Figure 14 shows the growth of the drsFTP receive buffer as a function of time during three transfers of a 512 MB file. The final buffer sizes for the three transfers range from 1.9 MB to 3.1 MB, with an average of 2.7 MB. Due to changing conditions during the transfers, the buffer sizes grow at different rates, particularly during the latter part of the transfer. In contrast, the over-provisioned FTP uses a 16 MB buffer which is statically allocated during connection set-up. Thus drsFTP achieves over three-quarters of the over-provisioned performance while only using one-sixth the amount of memory. In other words, drsFTP achieves an average of 10.1 Mbps per MB of buffer space used while statically-tuned FTP achieves only 2.2 Mbps per MB of buffer space used.

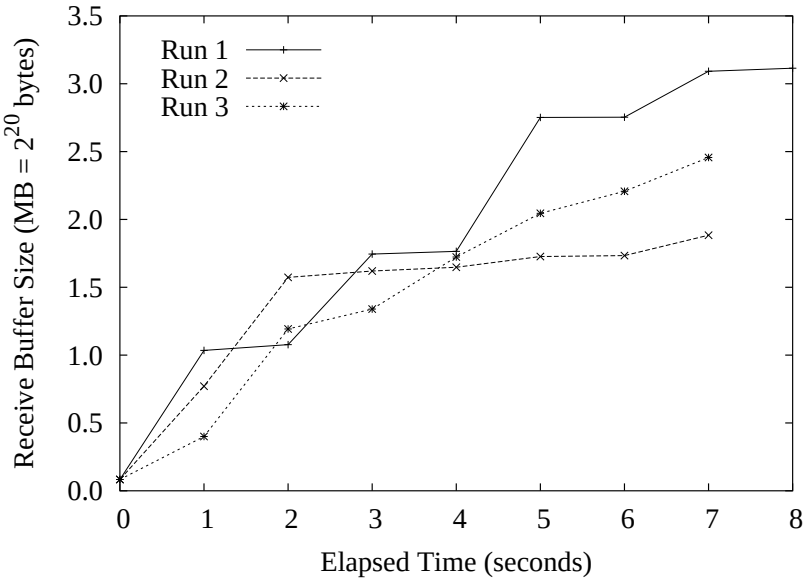

Figure 14. drsFTP Buffer Sizes over Time

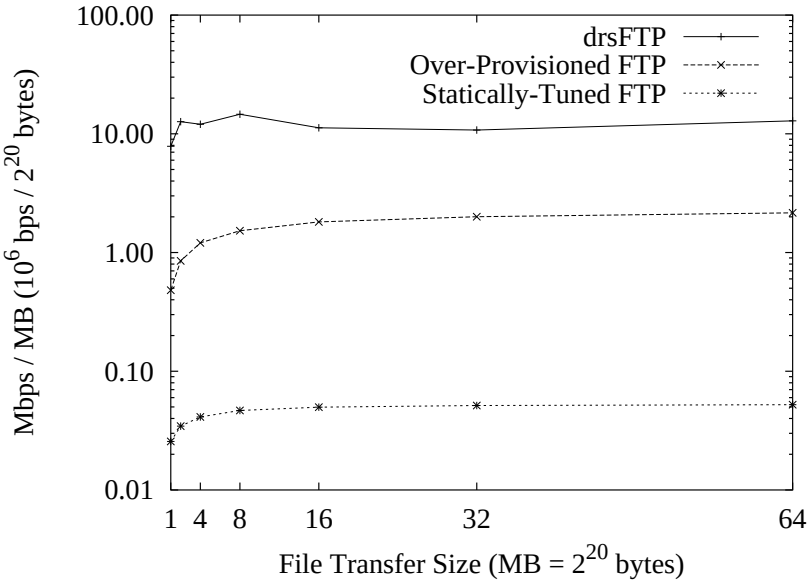

Figure 15. Mbps per MB of Buffer Space

As Figure 15 shows, drsFTP achieves five times better utilization of the network with respect to memory than the over-provisioned case. Even if the theoretically optimal BDP of 12.2 MB been allocated instead of over-provisioning, drsFTP would still have been able to support more connections with a 3.6 times improvement in Mbps per MB. The difference between drsFTP and the statically-tuned case where the BDP was sampled at an inopportune time is even more dramatic.

6. Future Work

The results of the experiments conducted so far indicate that DRS, both kernel- and user-space, are likely to perform well "in the wild". Anecdotally, we have observed very good performance on live networks but still need to rigorously quantify the improvements. We also need to do more testing on connections with low and medium bandwidth-delay products.

Next, DRS needs to be thoroughly studied in the context of parallel streams. Does DRS subsume the need for parallel streams or will a combined approach be best? Section 5.2.2 suggests that fewer DRS streams are needed than with regular TCP.

Finally, we are working to get DRS incorporated into the official Linux source tree. Once incorporated, applications will transparently see an increase in delivered bandwidth. In the mean time, we are continuing to develop drsFTP and to integrate drsFTP with GridFTP.

7. Conclusion

This paper makes three significant contributions to the high-speed networking and grid-computing communities. First, we demonstrated that the bandwidth-delay product (BDP) can vary widely over the lifetime of a connection. Therefore, simply tuning buffers at connection set-up is not good enough; they must be tuned over the lifetime of the connection. This is the motivation for Dynamic Right-Sizing (DRS).

Further, since we used `nettimer` to measure the BDP of a connection, our estimates may be conservative because `nettimer` measures *static* bottleneck bandwidth and dynamic delay. With the recent release of `pathload` [20], which measures dynamic available bandwidth and delay, our initial tests indicate that the BDP actually fluctuates by an additional order of magnitude.

Second, we illustrated how a receiver can measure the bandwidth and round-trip delay of a connection (i.e., BDP) *without* 'polluting' the network with any extraneous probing packets. This BDP value is then used as an upper bound for the flow-control window in DRS.

Third, in the context of DRS, we have shown how a TCP receiver can determine the approximate size of the sender's congestion window so that the receiver can advertise a flow-control window that neither needlessly constrains throughput nor unnecessarily over-allocates buffer space. Furthermore, this can be done automatically and transparently while abiding by TCP semantics. It can also be done in user space, although with a small reduction in performance.

Finally, we are making our implementations of DRS in the Linux 2.4 kernel and in user-space available under an open-source license. The DRS kernel implementation has already been incorporated into the Web100 project [11]. Furthermore, we are in the process of integrating drsFTP with GridFTP to make the benefits of Dynamic Right-Sizing available to the Globus community.

Notes

[1]The *wizard gap* is the difference between the network performance that a network 'wizard' can achieve by appropriately tuning buffer sizes and the performance of an untuned application.

[2]Because sender-side auto-tuning [35] ignores *fwnd*, it breaks TCP semantics by allowing the sender to overrun the receiver whether inadvertently (as in an FTP transfer) or maliciously (as in a denial-of-service attack).

[3]Although we would have liked to sample at the granularity of the RTT, the overhead of running *nettimer* and other tools in user space prevented us from obtaining the measurements we sought.

[4]DRS does assume that end hosts have a fair buffer allocation policy. If the buffer allocation policy is not fair, both regular TCP and DRS will be unfair. The fault lies with the buffer allocation policy not the transport protocol.

[5]We assume the end-hosts are not bottlenecks and hence it makes sense to seek higher bandwidth.

[6]Equivalently, we can select a fixed number of bytes to be received periodically and measure how long it takes.

References

1. 'NcFTP Software Client and Server'. `http://www.ncftp.com/`.
2. 'The Netkit FTP Client'. `http://freshmeat.net/projects/netkit/`.
3. 'The Particle Physics Data Grid'. `http://www.cacr.caltech.edu/ppdg/`.
4. 'The Washington University Archive FTP daemon (wu-ftpd)'. `http://www.wu-ftpd.org/`.
5. Allcock, W. et al.: 2001, 'GridFTP: Protocol Extensions to FTP for the Grid'. `http://www-fp.mcs.anl.gov/dsl/GridFTP-Protocol-RFC-Draft.pdf`.
6. Allman, M. et al.: 2000, 'Ongoing TCP Research Related to Satellites'. IETF RFC 2760.
7. Allman, M., D. Glover, and L. Sanchez: 1999, 'Enhancing TCP Over Satellite Channels Using Standard Mechanisms'. IETF RFC 2488.

8. Bansal, D., H. Balakrishnan, S. Floyd, and S. Shenker: 2001, 'Dynamic Behavior of Slowly-Responsive Congestion Control Algorithms'. In: *SIGCOMM 2001*. San Diego, CA.
9. Chervenak, A., I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke: 2001, 'The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Datasets'. *International Journal of Supercomputer Applications* **23**(3), 187–200.
10. Childers, L., T. Disz, R. Olson, M. E. Papka, R. Stevens, and T. Udeshi: 2000, 'Access Grid: Immersive Group-to-Group Collaborative Visualization'. In: *Proceedings of the 4th International Immersive Projection Workshop*.
11. Dunigan, T. and F. Fowler, 'Personal Communication with Web100 Project'. 2002.
12. Feng, W. and P. Tinnakornsrisuphap: 2000, 'The Failure of TCP in High-Performance Computational Grids'. In: *Proceedings of SC 2000: High-Performance Networking and Computing Conference*.
13. Fisk, M. and W. Feng: 2000, 'Dynamic Adjustment of TCP Window Sizes'. Technical Report Los Alamos Unclassified Report (LAUR) 00-3221, Los Alamos National Laboratory.
14. Fisk, M. and W. Feng: 2001, 'Dynamic Right-Sizing: TCP Flow-Control Adaptation'. In: *Proceedings of SC 2001: High-Performance Networking and Computing Conference*.
15. Floyd, S.: 2002, 'HighSpeed TCP for Large Congestion Windows'. `http://www.ietf.org/internet-drafts/draft-floyd-tcp-highspeed-01.txt`.
16. Foster, I. and C. Kesselman (eds.): 1998, *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers.
17. Foster, I., C. Kesselman, and S. Tuecke: 2001, 'The Anatomy of the Grid: Enabling Scalable Virtual Organizations'. *International Journal of Supercomputer Applications*.
18. Gardner, M., W. Feng, and M. Fisk: 2002, 'Dynamic Right-Sizing in FTP: Enhancing Grid Performance in User Space'. In: *Proceedings of the IEEE Symposium on High-Performance Distributed Computing*.
19. Jacobson, V., R. Braden, and D. Borman: 1992, 'TCP Extensions for High Performance'. IETF RFC 1323.
20. Jain, M. and C. Dovrolis: 2002, 'End-to-end Available Bandwidth: Measurement Methodology, Dynamics, and Relation with TCP Throughput'. In: *Proceedings of the Annual Conference of the Special Interest Group on Data Communication (SIGCOMM)*.
21. Jin, G., G. Yang, B. Crowley, and D. Agrawal: 2001, 'Network Characterization Service'. In: *Proceedings of the IEEE Symposium on High-Performance Distributed Computing*.
22. Kelly, T.: 2002, 'On Engineering a Stable and Scalable TCP Variant'. Technical Report CUED/F-INFENG/TR.435, Cambridge University Engineering Department. `http://www-lce.eng.cam.ac.uk/~ctk21/papers/sstcp-variant.pdf`.
23. Lai, K. and M. Baker: 2001, 'Nettimer: A Tool for Measuring Bottleneck Link Bandwidth'. In: *Proceedings of the USENIX Symposium on Internet Technologies and Systems*.
24. Lawrence Berkley National Laboratory, 'Nettest: Secure Network Testing and Monitoring'. `http://www-itg.lbl.gov/nettest/`.

25. Liu, J. and J. Ferguson: 2000, 'Automatic TCP Socket Buffer Tuning'. In: *Proceedings of SC 2000: High-Performance Networking and Computing Conference (Research Gem)*. http://dast.nlanr.net/Projects/Autobuf.
26. Mah, B., 'pchar: A Tool for Measuring Internet Path Characteristics'. http://www.employees.org/~bmah/Software/pchar.
27. Mathis, M., 'Pushing Up Performance for Everyone'. http://www.ncne.nlanr.net/news/workshop/19999/991205/Talks/mathis_991205_Pushing_Up_Performance/.
28. Muuss, M. J., 'The TTCP Program'. http://ftp.arl.mil/~mike/ttcp.html.
29. National Center for Atmospheric Research and Pittsburgh Supercomputing Center and National Center for Supercomputing Applications, 'The Web100 Project'. http://www.web100.org/.
30. National Laboratory for Applied Network Research, 'Automatic TCP Window Tuning and Applications'. http://dast.nlanr.net/Projects/Autobuf_v1.0/autotcp.html.
31. Partridge, C. and T. Shepard: 1997, 'TCP/IP Performance over Satellite Links'. *IEEE Network*.
32. Pittsburgh Supercomputing Center, 'Enabling High-Performance Data Transfers on Hosts'. http://www.psc.edu/networking-/perf_tune.html/.
33. Pittsburgh Supercomputing Center, 'Enabling High Performance Data Transfers on Hosts'. http://www.psc.edu/networking/perf_tune.html.
34. Postel, J. and J. Reynolds: 1995, 'File Transfer Protocol (FTP)'. IETF RFC 959.
35. Semke, J., J. Mahdavi, and M. Mathis: 1998, 'Automatic TCP Buffer Tuning'. *Computer Communications Review, ACM SIGCOMM* **28**(4).
36. Sivakumar, H., S. Bailey, and R. L. Grossman: 2000, 'PSockets: The Case for Application-level Network Striping for Data Intensive Applications using High Speed Wide Area Networks'. In: *Supercomputing*.
37. Tierney, B.: 2001, 'TCP Tuning Guide for Distributed Applications on Wide-Area Networks'. In: *USENIX & SAGE Login*. http://www-didc.lbl.gov/tcp-wan.html.
38. Tierney, B., D. Gunter, J. Lee, and M. Stoufer: 2001, 'Enabling Network-Aware Applications'. In: *Proceedings of the IEEE International Symposium on High-Performance Distributed Computing*.
39. Tirumala, A. and J. Ferguson: 2001, 'IPERF'. http://dast.nlanr.net/Projects/Iperf/index.html.
40. University of Kansas, Information & Telecommunication Technology Center, 'NetSpec: A Tool for Network Experimentation and Measurement'. http://www.ittc.ukans.edu/netspec/.
41. Weigle, E. and W. Feng: 2001, 'Dynamic Right-Sizing: A Simulation Study'. In: *Proceedings of IEEE International Conference on Computer Communications and Networks*. http://public.lanl.gov/ehw/papers/ICCCN-2001-DRS.ps.
42. Weigle, E. and W. Feng: 2002a, 'A Comparison of TCP Automatic Tuning Techniques for Distributed Computing'. In: *Proceedings of the IEEE Symposium on High-Performance Distributed Computing (HPDC-11)*.
43. Weigle, E. and W. Feng: 2002b, 'TICKETing High-Speed Traffic with Commodity Hardware and Software'. In: *Proceedings of the Third Annual Passive and Active Measurement Workshop (PAM2002)*.